OSD Practical-12
Pre-Lab

1. **Creating and Destroying Condition Variables: pthread_cond_init (condition,attr) pthread_cond_destroy (condition)**

Ans:

The **pthread_cond_destroy()** function shall destroy the given condition variable specified by cond; the object becomes, in effect, uninitialized. An implementation may cause pthread_cond_destroy() to set the object referenced by cond to an invalid value. A destroyed condition variable object can be reinitialized using pthread_cond_init(); the results of otherwise referencing the object after it has been destroyed are undefined.
It shall be safe to destroy an initialized condition variable upon which no threads are currently blocked. Attempting to destroy a condition variable upon which other threads are currently blocked results in undefined behavior.

The **pthread_cond_init()** function shall initialize the condition variable referenced by cond with attributes referenced by attr. If attr is NULL, the default condition variable attributes shall be used; the effect is the same as passing the address of a default condition variable attributes object. Upon successful initialization, the state of the condition variable shall become initialized.
Only cond itself may be used for performing synchronization. The result of referring to copies of cond in calls to pthread_cond_wait(), pthread_cond_timedwait(), pthread_cond_signal(), pthread_cond_broadcast(), and pthread_cond_destroy() is undefined.
Attempting to initialize an already initialized condition variable results in undefined behavior.
In cases where default condition variable attributes are appropriate, the macro PTHREAD_COND_INITIALIZER can be used to initialize condition variables that are statically allocated. The effect shall be equivalent to dynamic initialization by a call to pthread_cond_init() with parameter attr specified as NULL, except that no error checks are performed

2. **Waiting and Signaling on Condition Variables: pthread_cond_wait (condition,mutex) pthread_cond_signal (condition) pthread_cond_broadcast (condition)**

Ans:

pthread_cond_wait Syntax

```
int pthread_cond_wait(pthread_cond_t *restrict cv,pthread_mutex_t *restrict mutex);
```

```
#include <pthread.h>
pthread_cond_t cv;
pthread_mutex_t mp;
int ret;
/* wait on condition variable */
ret = pthread_cond_wait(&cv, &mp);
```

The blocked thread can be awakened by a pthread_cond_signal() , a pthread_cond_broadcast() , or when interrupted by delivery of a signal.
Any change in the value of a condition that is associated with the condition variable cannot be inferred by the return of pthread_cond_wait(). Such conditions must be reevaluated.
The pthread_cond_wait() routine always returns with the mutex locked and owned by the calling thread, even when returning an error.
This function blocks until the condition is signaled. The function atomically releases the associated mutex lock before blocking, and atomically acquires the mutex again before returning.
In typical use, a condition expression is evaluated under the protection of a mutex lock. When the condition expression is false, the thread blocks on the condition variable. The condition variable is then signaled by another thread when the thread changes the condition value. The change causes at least one thread that is waiting on the condition variable to unblock and to reacquire the mutex.
The condition that caused the wait must be retested before continuing execution from the point of the pthread_cond_wait(). The condition could change before an awakened thread reacquires the mutes and returns from pthread_cond_wait(). A waiting thread could be awakened spuriously. The recommended test method is to write the condition check as a while() loop that calls pthread_cond_wait().

```
  pthread_mutex_lock();
     while(condition_is_false)
        pthread_cond_wait();
  pthread_mutex_unlock();
```

The scheduling policy determines the order in which blocked threads are awakened. The default scheduling policy, SCHED_OTHER, does not specify the order in which threads are awakened. Under the SCHED_FIFO and SCHED_RR real-time scheduling policies, threads are awakened in priority order.

pthread_cond_signal Syntax

```
int pthread_cond_signal(pthread_cond_t *cv);
#include <pthread.h>
pthread_cond_t cv;
int ret;

/* one condition variable is signaled */
ret = pthread_cond_signal(&cv);
```

Modify the associated condition under the protection of the same mutex used with the condition variable being signaled. Otherwise, the condition could be modified between its test and blocking in pthread_cond_wait(), which can cause an infinite wait.
The scheduling policy determines the order in which blocked threads are awakened. The default scheduling policy, SCHED_OTHER, does not specify the order in which threads are awakened. Under the SCHED_FIFO and SCHED_RR real-time scheduling policies, threads are awakened in priority order.
When no threads are blocked on the condition variable, calling pthread_cond_signal() has no effect.

pthread_cond_broadcast Syntax

int pthread_cond_broadcast(pthread_cond_t *cv);

```
#include <pthread.h>
pthread_cond_t cv;
int ret;

/* all condition variables are signaled */
ret = pthread_cond_broadcast(&cv);
```

When no threads are blocked on the condition variable, pthread_cond_broadcast() has no effect.

Since pthread_cond_broadcast() causes all threads blocked on the condition to contend again for the mutex lock, use pthread_cond_broadcast() with care. For example, use pthread_cond_broadcast() to allow threads to contend for varying resource amounts when resources are freed

INLAB

1. Solve producer consumer problem using mutex, binary and counting semaphores, and condition variables.

Ans:

```
/* prodcons-mutex.c - Producer Consumer problem using mutex and pthreads */
/* include main */
#include	<stdio.h>
#include	<unistd.h>
#include	<fcntl.h>
#include	<pthread.h>
#include	<sys/types.h>
#define	MAXNITEMS		1000000
#define	MAXNTHREADS			100

int		nitems;			/* read-only by producer and consumer */
struct {
  pthread_mutex_t	mutex;
  int	buff[MAXNITEMS];
  int	nput;
  int	nval;
} shared ={PTHREAD_MUTEX_INITIALIZER};

void	*produce(void *), *consume(void *);

int
main(int argc, char **argv)
{
	shared.nput=0;
	shared.nval=0;
	int			i, nthreads, count[MAXNTHREADS];
	pthread_t	tid_produce[MAXNTHREADS], tid_consume;

	if (argc != 3)
		printf("usage: prodcons1 <#items> <#threads>");
	nitems = atoi(argv[1]);
	nthreads = atoi(argv[2]);
	pthread_setconcurrency(nthreads);
		/* 4start all the producer threads */
	for (i = 0; i < nthreads; i++) {
		count[i] = 0;
		pthread_create(&tid_produce[i], NULL, produce, &count[i]);
	}

		/* 4wait for all the producer threads */
	for (i = 0; i < nthreads; i++) {
		pthread_join(tid_produce[i], NULL);
```

```
                    printf("count[%d] = %d\n", i, count[i]);
        }

                    /* 4start, then wait for the consumer thread */
        pthread_create(&tid_consume, NULL, consume, NULL);
        pthread_join(tid_consume, NULL);

        exit(0);
}
/* end main */

/* include producer */
void *
produce(void *arg)
{
pthread_t tid;
int i=*((int *) arg);
        for ( ; ; ) {
                    pthread_mutex_lock(&shared.mutex);
                    tid=pthread_self();
                    printf("threadid=%u\n", (unsigned int) tid);
                    if (shared.nput >= nitems) {
                                pthread_mutex_unlock(&shared.mutex);
                                return(NULL);           /* array is full, we're done */
                    }
                    shared.buff[shared.nput] = shared.nval;
                    printf("buff[%d] = %d\n", shared.nput, shared.buff[shared.nput]);
                    shared.nput++;
                    shared.nval++;
                    *((int *) arg) += 1;
                    pthread_mutex_unlock(&shared.mutex);
                    printf("shared.nput=%d, shared.nval=%d,count[%u] =
%d\n",shared.nput,shared.nval, i, *((int *) arg));

        }
}

void * consume(void *arg)
{
        int     i;
        for (i = 0; i <nitems; i++) {
                    if (shared.buff[i] != i)
                                printf("buff[%d] = %d\n", i, shared.buff[i]);
        }
        return(NULL);
}
/* end producer */
```

osd-190031187@team-osd:~

```
[osd-190031187@team-osd ~]$ cc prodcons-mutex.c -lpthread
[osd-190031187@team-osd ~]$ ./a.out 5 4
threadid=2423367424
buff[0] = 0
shared.nput=1, shared.nval=1,count[0] = 1
threadid=2423367424
buff[1] = 1
shared.nput=2, shared.nval=2,count[0] = 2
threadid=2423367424
buff[2] = 2
shared.nput=3, shared.nval=3,count[0] = 3
threadid=2406582016
buff[3] = 3
threadid=2398189312
buff[4] = 4
shared.nput=5, shared.nval=5,count[0] = 1
threadid=2398189312
threadid=2414974720
threadid=2423367424
count[0] = 3
count[1] = 0
shared.nput=4, shared.nval=4,count[0] = 1
threadid=2406582016
count[2] = 1
count[3] = 1
[osd-190031187@team-osd ~]$
```

2. prodcons-cv.c

```
/* Implementation of Producer consumer problem using condition variables */
/* include globals */
#include        <stdio.h>
#include        <unistd.h>
#include        <fcntl.h>
#include        <pthread.h>
#include        <sys/types.h>

#define         MAXNITEMS           1000000
#define         MAXNTHREADS                 100

                /* globals shared by threads */
int             nitems;                             /* read-only by producer and consumer
*/
int             buff[MAXNITEMS];
struct {
  pthread_mutex_t      mutex;
  int                          nput;   /* next index to store */
```

```
  int                               nval;    /* next value to store */
} put = { PTHREAD_MUTEX_INITIALIZER };

struct {
  pthread_mutex_t      mutex;
  pthread_cond_t       cond;
  int                               nready;          /* number ready for consumer */
} nready = { PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER };
/* end globals */

void     *produce(void *), *consume(void *);

/* include main */
int
main(int argc, char **argv)
{
        int                          i, nthreads, count[MAXNTHREADS];
        pthread_t        tid_produce[MAXNTHREADS], tid_consume;

        if (argc != 3)
        {
                printf("usage: prodcons6 <#items> <#threads>");
        }
        nitems = atoi(argv[1]);
        nthreads = atoi(argv[2]);

        pthread_setconcurrency(nthreads + 1);
                /* 4create all producers and one consumer */
        for (i = 0; i < nthreads; i++) {
                count[i] = 0;
                pthread_create(&tid_produce[i], NULL, produce, &count[i]);
        }
        pthread_create(&tid_consume, NULL, consume, NULL);

                /* wait for all producers and the consumer */
        for (i = 0; i < nthreads; i++) {
                pthread_join(tid_produce[i], NULL);
                printf("count[%d] = %d\n", i, count[i]);
        }
        pthread_join(tid_consume, NULL);

        exit(0);
}
/* end main */

/* include prodcons */
void *
```

```
produce(void *arg)
{
	for ( ; ; ) {
		pthread_mutex_lock(&put.mutex);
		if (put.nput >= nitems) {
			pthread_mutex_unlock(&put.mutex);
			return(NULL);		/* array is full, we're done */
		}
		buff[put.nput] = put.nval;
		put.nput++;
		put.nval++;
		pthread_mutex_unlock(&put.mutex);

		pthread_mutex_lock(&nready.mutex);
		if (nready.nready == 0)
			pthread_cond_signal(&nready.cond);
		nready.nready++;
		pthread_mutex_unlock(&nready.mutex);

		*((int *) arg) += 1;
	}
}

void *
consume(void *arg)
{
	int		i;

	for (i = 0; i < nitems; i++) {
		pthread_mutex_lock(&nready.mutex);
		while (nready.nready == 0)
			pthread_cond_wait(&nready.cond, &nready.mutex);
		nready.nready--;
		pthread_mutex_unlock(&nready.mutex);

		if (buff[i] == i)
			printf("buff[%d] = %d\n", i, buff[i]);
	}
	return(NULL);
}
/* end prodcons */
```

osd-190031187@team-osd:~

```
[osd-190031187@team-osd ~]$ cc prodcons-cv.c -lpthread
[osd-190031187@team-osd ~]$ ./a.out 5 4
count[0] = 5
count[1] = 0
count[2] = 0
count[3] = 0
buff[0] = 0
buff[1] = 1
buff[2] = 2
buff[3] = 3
buff[4] = 4
[osd-190031187@team-osd ~]$
```

POSTLAB

Solve readers writer's problem using mutex and semaphores
Ans:

```
#include<pthread.h>
        #include <semaphore.h>
        #include <stdio.h>


        /*
        This program provides a possible solution for first readers writers problem using
mutex and semaphore.
        I have used 10 readers and 5 producers to demonstrate the solution. You can always
play with these values.
        */


        sem_t wrt;
        pthread_mutex_t mutex;
        int cnt = 1;
        int numreader = 0;


        void *writer(void *wno)
        {
          sem_wait(&wrt);
          cnt = cnt*2;
          printf("Writer %d modified cnt to %d\n",(*((int *)wno)),cnt);
          sem_post(&wrt);


        }
        void *reader(void *rno)
        {
          // Reader acquire the lock before modifying numreader
          pthread_mutex_lock(&mutex);
          numreader++;
          if(numreader == 1) {
            sem_wait(&wrt); // If this id the first reader, then it will block the writer
          }
          pthread_mutex_unlock(&mutex);
          // Reading Section
          printf("Reader %d: read cnt as %d\n",*((int *)rno),cnt);


          // Reader acquire the lock before modifying numreader
          pthread_mutex_lock(&mutex);
```

```
        numreader--;
        if(numreader == 0) {
            sem_post(&wrt); // If this is the last reader, it will wake up the writer.
        }
        pthread_mutex_unlock(&mutex);
    }


    int main()
    {


        pthread_t read[10],write[5];
        pthread_mutex_init(&mutex, NULL);
        sem_init(&wrt,0,1);


        int a[10] = {1,2,3,4,5,6,7,8,9,10}; //Just used for numbering the producer and
consumer


        for(int i = 0; i < 10; i++) {
            pthread_create(&read[i], NULL, (void *)reader, (void *)&a[i]);
        }
        for(int i = 0; i < 5; i++) {
            pthread_create(&write[i], NULL, (void *)writer, (void *)&a[i]);
        }


        for(int i = 0; i < 10; i++) {
            pthread_join(read[i], NULL);
        }
        for(int i = 0; i < 5; i++) {
            pthread_join(write[i], NULL);
        }


        pthread_mutex_destroy(&mutex);
        sem_destroy(&wrt);


        return 0;

    }
```

osd-190031187@team-osd:~

```
[osd-190031187@team-osd ~]$ cc reader-writer.c -lpthread
[osd-190031187@team-osd ~]$ ./a.out
Reader 1: read cnt as 1
Reader 2: read cnt as 1
Reader 3: read cnt as 1
Reader 4: read cnt as 1
Reader 5: read cnt as 1
Reader 6: read cnt as 1
Reader 7: read cnt as 1
Reader 8: read cnt as 1
Reader 9: read cnt as 1
Reader 10: read cnt as 1
Writer 1 modified cnt to 2
Writer 2 modified cnt to 4
Writer 3 modified cnt to 8
Writer 4 modified cnt to 16
Writer 5 modified cnt to 32
[osd-190031187@team-osd ~]$
```